Querying Big Graphs Within Bounded Resources

Querying Big Graphs within Bounded Resources

Wenfei Fan[1,2] Xin Wang[4] Yinghui Wu[3]
[1]University of Edinburgh [2] RCBD & SKLSDE Lab, Beihang University [3]UC Santa Barbara
[4] Southwest Jiaotong University
{wenfei@inf, x.wang-36@sms, y.wu-18@sms}.ed.ac.uk

ABSTRACT

This paper studies the problem of querying graphs within bounded resources. Given a query Q, a graph G and a small ratio α, it aims to answer Q in G by accessing only a fraction G_Q of G of size $|G_Q| \leq \alpha|G|$. The need for this is evident when G is big while our available resources are limited, as indicated by α. We propose resource-bounded query answering via a dynamic scheme that reduces big G to G_Q. We investigate when we can find the exact answers $Q(G)$ from G_Q, and if G_Q cannot accommodate enough information, how accurate the approximate answers $Q(G_Q)$ are. To verify the effectiveness of the approach, we study two types of queries. One consists of pattern queries that have data locality, such as subgraph isomorphism and strong simulation. The other is the class of reachability queries, without data locality. We show that it is hard to get resource-bounded algorithms with 100% accuracy: NP-hard for pattern queries, and non-existing for reachability when $\alpha \neq 1$. Despite these, we develop resource-bounded algorithms for answering these queries. Using real-life and synthetic data, we experimentally evaluate the performance of the algorithms. We find that they scale well for both types of queries, and our approximate answers are accurate, even 100% for small α.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

bounded resource; graph querying; pattern matching

1. INTRODUCTION

Real-life graphs introduce challenges to query answering. (1) Such graphs are typically big. For instance, Facebook has 1 billion nodes and 140 billion links in its social graph[1], and a Web-scale graph is easily of PB size [17]. (2) Queries are routinely posed on these graphs, such as graph pattern queries [20, 33] and reachability queries [36]. Such queries are expensive. For a graph $G = (V, E)$ and a query Q, it takes $O(|V| + |E|)$ time when Q is to test whether one node can reach another in G, $O(|Q||V|(|V| + |E|))$ time to find matches of Q in G when Q is a graph pattern and matching is defined by strong simulation [20], and worse yet, it is NP-hard even to decide whether there exists a match of Q in G by subgraph isomorphism. It is often cost-prohibitive to find exact answers to these queries in big graphs.

Can we still answer such queries Q in a big graph G when we have limited resources, *e.g.,* time and space? This question motivates us to study *resource-bounded query answering*. Given a small ratio $\alpha \in (0, 1)$ and Q posed on G, we extract a fraction G_Q of G such that $|G_Q| \leq \alpha|G|$, and compute *approximate answers* $Q(G_Q)$. Here α is called a *resource ratio* and is determined by our available resources.

The idea behind resource-bounded query answering is to *make big data "small"*. While we cannot lower the complexity of computing $Q(G)$, we reduce the cost by using small G_Q instead of G, and hence, make it *feasible* to answer expensive queries in big graphs. The need for this is evident: real-life searches require fast response (*e.g.,* in less than 1 second [6]) with *e.g.,* limited memory [18] and energy [15]. Computation of exact answers $Q(G)$ by accessing the entire G is often beyond reach in these settings. We may have to settle with approximate answers, which often suffice in, *e.g.,* updating ads based on trends in social networks [2] and mining patterns in social graphs [9,16]. Moreover, for graph pattern matching by subgraph isomorphism, it is often necessary to adopt inexact query answers anyway.

Obviously the smaller the resource ratio α is, the less *space and time* it takes to compute $Q(G_Q)$; but as a price, the lower the accuracy of the approximate answers $Q(G_Q)$ is. Nonetheless, we show that even with small α, we can often find answers with quite good quality, even exact answers (Section 6). Indeed, a typical Facebook Graph Search query[2] can be answered by using nodes that are within 3 hops of a designated node in G, a small fraction of its entire social graph [34]. This is also the case for a range of personalized social search queries [1,8]. However, note that simply extracting local information alone may not suffice: there could be more than 10^9 nodes within 3 hops of a node [1].

Example 1: A fraction of a social network G is shown in Fig. 1. There are three social groups in G: a hiking group (HG), a city cycling club (CC), and a separate group of

[1]http://newsroom.fb.com/

[2]https://www.facebook.com/about/graphsearch

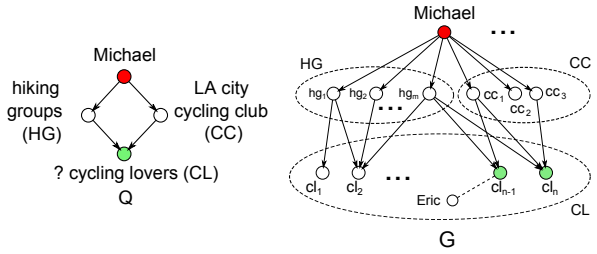


Figure 1: Personalized social search

cycling lovers (CL). A user Michael issues a query: "find me cycling lovers (CL) who know both my friends in LA cycling club (CC), and my friends in the hiking group (HG)". Such a query is common in, *e.g.,* Facebook Graph Search. It can be represented as graph pattern Q shown in Fig. 1.

By strong simulation [20], $Q(G)$ returns two matches cl_{n-1} and cl_n. To find them, there is no need to search the entire G. It suffices to consider a subgraph G_Q of G consisting of only those nodes within 2 hops of *Michael*. In fact, a small G_Q with 7 nodes will do here. Resource-bounded query answering aims to identify such a small subgraph G_Q of G within resource ratio α such that $Q(G_Q)$ can still give us accurate or even exact answers. Note that if the resource ratio α allows us to visit, say, at most 16 nodes and edges in G_Q, it is *nontrivial* to identify a sensible G_Q from (possibly thousands of) nodes within 2 hops of *Michael*.

The query Q is *localized*: one can find matches locally, by searching only those nodes within d_Q hops of a designated note *Michael*, where d_Q is decided by Q. Thus to answer Q we do not have to consider nodes far from *Michael*.

More challenging are *non-localized* queries. For example, Michael also asks whether he can reach a sport star Eric via social links. This reachability query is non-localized: in the worst case, every node and edge in G may have to be visited. We will show that we can still answer such queries rather accurately without traversing the entire G. □

This example shows that resource-bounded query answering is feasible in practice, and it helps us query big data. To make practical use of it, however, several questions have to be answered. Given a ratio α, how can we identify a small G_Q that is within the bound *w.r.t.* α, and gives us accurate or approximate answers to Q in G? Can we get accuracy guarantees in G_Q, even 100% accurate? Does this approach work on both localized queries and non-localized queries? These questions are challenging even for localized queries.

Contribution. This work is a step towards effective approaches to answering queries within bounded resources.

(1) We formalize resource-bounded query answering via a dynamic reduction scheme (Section 3). Given a graph G, a query Q and a small $\alpha \in (0, 1)$, we propose to first reduce G to G_Q such that $|G_Q| \leq \alpha|G|$, by accessing a bounded amount of the data in G. We then compute $Q(G_Q)$ as approximate answers to Q in G. We define query answer accuracy to evaluate the quality of approximate answers $Q(G_Q)$.

As a proof of concept, we study resource-bounded query answering for both localized and non-localized queries.

(2) We develop resource-bounded algorithms for graph pattern matching in terms of strong simulation and subgraph isomorphism (Section 4). We show that for these localized queries, resource-bounded query answering is already nontrivial. It is NP-hard to decide, given G, Q and α, whether there exists a subgraph G_Q of G such that $|G_Q| \leq \alpha|G|$ and $Q(G) = Q(G_Q)$, *i.e.,* whether it is possible to find exact answers from a small G_Q. Despite this, we develop a reduction strategy that identifies G_Q by fetching nodes based on their dynamically maintained weights, guided by query Q. We show that our algorithms visit a bounded amount of data in G, and possess certain accuracy guarantees.

(3) We extend the study to reachability queries (Section 5). We show that for these non-localized queries, there exist *no* algorithms that, given G, nodes s and t in G, and $\alpha < 1$, decide whether s reaches t with 100% accuracy by *visiting no more than* $\alpha|G|$ *nodes*. While reachability can be tested in linear time, it has to visit *an unbounded number* of nodes in G or store an index larger than G. To this end, we provide an algorithm for answering reachability queries based on a hierarchical landmark index. Using the index, the algorithm drills-down or rolls-up in the search, visits at most $\alpha|G|$ nodes or edges, and *guarantees 100% true positives*.

(4) We experimentally evaluate the effectiveness of the approach using real-life and synthetic graphs (Section 6). We find that our algorithms are (a) *efficient*: they are 5.5, 6.25 and 5.7 times faster than traditional algorithms for strong simulation, subgraph isomorphism and reachability, respectively, even after they are improved by employing *our own optimization*; (b) *accurate*: they often achieve 100% accuracy by accessing only 0.0015% (resp. 0.05%) of graphs G to answer pattern (resp. reachability) queries; for pattern queries, they visit 7%-24% of the data in the neighborhood of a personalized node within $|Q|$ hops; and (c) *scalable*: they scale well when G grows; *e.g.,* for $\alpha = 0.0015\%$ (*i.e.,* $15 * 10^{-6}$) and $|G| = 1$ PB, they access only 15GB of data, *reducing G from PB to GB* while retaining high accuracy.

We contend that resource-bounded query answering is capable of finding accurate answers by accessing a small fraction of big graphs, and is promising for evaluating both localized and non-localized queries in real life. This also suggests how we can *strike a balance* between the resources needed and the accuracy of approximate answers computed.

Related work. We categorize the related work as follows.

Indexing and compression. There are typically two ways to reduce the search space: *indexing* and *compression*.

(1) Graph indexing [10, 13, 26, 35] provides precomputed global information of G to evaluate queries, with additional storage costs. For instance, given $G = (V, E)$ for reachability queries, a reachability matrix takes $O(|V|^2)$ space to store [36]. A 2-hop index takes $O(|V||E|^{\frac{1}{2}})$ space to store and $O(|E|^{\frac{1}{2}})$ time to query. These are not very practical when G is big. Labeling-based methods for reachability queries are studied in [35] with reduced index size by pruning landmark and path labeling. In contrast, this work uses small indices to support dynamic reduction, while striking a balance between the amount of data accessed (bounded by a *given small ratio*) and the accuracy of query results.

(2) Graph compression [4, 14] constructs a summary of G. To answer Q, however, it often needs decompression, sometimes restoring the entire G [4]. In a similar sense, graph summarization gives sketches of G [23,32]. Query preserving compression [12] allows us to process Q without decompression. It compresses G into a graph G_c (5% and 43% of $|G|$ for reachability and graph simulation, respectively).

For Web-scale graphs of PB size, however, this technique alone does not suffice. In contrast to compression that uses *the same* G_c to answer *all queries* posed on G, we fetch a bounded G_Q given *each* query Q with information for answering the particular Q. This said, the technique of [12] can be seamlessly combined with ours as a preprocessing step.

Distributed systems. Distributed systems, *e.g.,* Pregel [21] and GraphLab [19], evaluate queries on vertices of a graph in parallel with multiple processors. In contrast, this work studies query evaluation with limited resources and a single processor. This said, the techniques of this work can be readily adapted to the distributed settings.

Budgeted search. Related is also prior work on finding error-bounded answers, as early as (weighted) A^* [25], which was recently extended as optimistic search [30]. The prior work focuses on predicating how good a partial answer (as in a search tree) approximates the optimal solution, but the cost of finding such answers is not the major concern. Bounded-cost search was recently proposed [29,31] for planning, with the cost bounded by a user-specified budget. The quality of the answer, however, is not a concern [29]. In contrast, we aim to strike a balance between the cost of finding solutions and the quality of the answers, via dynamic data reduction.

Budgeted strategies for graph search were studied for *e.g.,* subgraph isomorphism [5, 28]. The idea of [5] is to assign dynamically maintained budgets and costs to nodes during the traversal, to find exact answers with minimal search space. To reduce verification cost, [28] schedules search order based on the frequencies of features in queries and data graphs. For graph patterns, our dynamic reduction is in a similar spirit, to greedily select promising nodes that may contribute to query answers. The difference is that we aim to process queries within *a given* (arbitrarily small) ratio α on the search space. Moreover, we provide methods to assess promising nodes and to guarantee bounded search space.

Closer to our work is BlinkDB [2] for relational queries. It adaptively samples data to find approximate query answers. "Predictable" queries are studied where enough information, *e.g.,* query logs and trace, is known to enable efficient precomputation of samples. In contrast, we study graph pattern queries, where sampling is much harder. This is because (1) the graph queries are rather "unpredictable" [2] due to flexible predicates posed on query nodes, and (2) in contrast to homogeneous table data, there is no "one-fit-for-all" schema available for data nodes in a graph. We also do not assume the existence of abundant query logs and workload for sampling strategy. Instead, we develop dynamic reduction techniques to identify and only access promising "areas" that lead to reasonable approximate answers.

2. PRELIMINARY

In this section we present localized queries and non-localized queries. We first review several basic notations.

Data graphs. We define a data graph as a node-labeled, directed graph $G = (V, E, L)$, where (1) V is a finite set of data nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; and (3) for each node v in V, $L(v)$ is the *label* of v. The label $L(v)$ may indicate *e.g.,* the content of a page [3] or node attributes [27].

We use two types of subgraphs $G_s = (V_s, E_s, L_s)$ of G.

- Graph G_s is a *subgraph of* G if $V_s \subseteq V$, $E_s \subseteq E$, and E_s (resp. L_s) is the restriction of E (resp. L) on the nodes in V_s; *i.e.,* for each edge $e = (v, v') \in E_s$, $v \in V_s$ and $v' \in V_s$; and for each $v \in V_s$, $L_s(v) = L(v)$.
- Graph G_s is a *subgraph of* G *induced by* V_s if it is a subgraph of G_s and for all nodes $v, v' \in V_s$, edge $(v, v') \in E_s$ if and only if $(v, v') \in E$; *i.e., E_s* includes *all* the edge of E that are defined on the nodes in V_s.

We will also use the following notations. (1) The *size* of a graph G, denoted as $|G|$, is the total number of the nodes and edges of G. We also use $|V|$ to denote the number of nodes in G; similarly for $|E|$. (2) The *diameter* of G is the length of the longest shortest path between any two nodes in G. (3) We say that a node v' is *within r hops* of v if there exists a path of at most r edges from v to v' or from v' to v. We denote by $N_r(v)$ the set of all nodes in G within r hops of v. (4) For a node v and a non-negative integer r, the *r-neighborhood* $G_r(v)$ of v is the subgraph of G induced by $N_r(v)$. (5) We say that v is a *parent* of v', or equivalently, v' is a *child* of v, if (v, v') is an edge in E.

We study two types of graph queries, given as follows.

Graph pattern queries. We study graph patterns for *personalized social search* [7,8]. A *graph pattern* is a graph $Q = (V_p, E_p, f_v, u_p, u_o)$, where (1) V_p and E_p are the set of query nodes and (directed) edges, respectively; (2) for each node u, $f_v(u)$ specifies a node label; and (3) u_p and u_o represent the *personalized node* and *output node* of Q, respectively.

In a data graph G, *the personalized node u_p has a unique match v_p*, with $f_v(u_p) = L(v_p)$, often denoting the person who issues the query Q. The output node u_o indicates the search intent of Q, and the label $f_v(u)$ specifies search constraints [7]. For instance, for the graph pattern Q over graph G of Fig. 1, node Michael is its personalized node, and has a unique match *Michael* in G. Node CL is the output node, indicating that the query is to find and return cycling lovers who satisfy the constraints of the pattern.

We consider two semantics for matching a graph pattern $Q = (V_p, E_p, f_v, u_p, u_o)$ to a data graph G.

Subgraph queries. A *match* of Q in G via *subgraph isomorphism* is a subgraph G' of G that is isomorphic to Q, *i.e.,* there exists a *bijective function h* from V_p to the set of nodes of G' such that (1) for each node $u \in V_p$, $f_v(u) = L(v)$; (2) (u, u') is an edge in Q if and only if $(h(u), h(u'))$ is an edge in G'; and (3) $h(u_p) = v_p$, *i.e., u_p* matches the *unique* v_p.

The *answer to Q in G*, denoted by $Q(G)$, is the set of nodes $h(u_o)$ that match the output node u_o of Q in G', for *all matches G'* of Q in G. We refer to Q as a *subgraph query*.

Simulation queries. For matching by strong simulation [20], a match of pattern Q in G is defined on the d_Q-neighborhood $G_{d_Q}(v_0) = (V_{d_Q}, E_{d_Q}, L_{d_Q})$ of nodes v_0 in G, where d_Q is the diameter of Q. In this setting, we say that *G matches Q* if there exists a binary relation $R_{v_0} \subseteq V_p \times V_{d_Q}$ such that

- $(u_p, v_p) \in R_{v_0}$, *i.e.,* the match of u_p is fixed to be v_p;
- for each node $u \in V_p$, there exists a *node* $v \in V_{d_Q}$ such that $(u, v) \in R_{v_0}$, referred to a *match* of u; and
- for each pair $(u, v) \in R_{v_0}$, $f_v(u) = L_{d_Q}(v)$ and further,
  - (a) for each edge (u, u') in E_p, there exists an edge $(v, v') \in E_{d_Q}$, such that $(u', v') \in R_{v_0}$, and
  - (b) for each edge (u'', u) in E_p, there exists an edge $(v'', v) \in E_{d_Q}$, such that $(u'', v'') \in R_{v_0}$.

Conditions (a) and (b) above ensure that the match preserves the children and parent relationships, respectively.

symbols	notations
$N_r(v)$	node set within r hops of v
$G_r(v)$	r-neighborhood graph of v
Q	$(V_p, E_p, f_v, u_p, u_o)$, graph pattern
u_p (resp. u_o)	personalized (resp. output) node in Q
v_p	the unique node in G that matches u_p
d_Q	the diameter of Q
l	the number of distinct labels in Q
d	the diameter of Q as an undirected graph
α	resource ratio such that $\|G_Q\| \leq \alpha\|G\|$
η	accuracy ratio: $\mathsf{accuracy}(Q, G, Q(G_Q)) \geq \eta$
f	the max number of nodes in $G_{d_Q}(v_p)$ sharing the same label and a common parent or child

Table 1: Notations: graphs and queries

The *match relation* R of Q in G is defined as the union of R_{v_0} *for all nodes* v_0 *in* G. For any Q and G, it is known that there exists a *unique, maximum* match relation R_M via strong simulation [20]. We define the *answer* $Q(G)$ to Q in G to be the set of matches of the output node u_o, *i.e.,* $Q(G) = \{v \mid (u_o, v) \in R_M\}$. We refer to Q as a *simulation query.*

For instance, for Q and G depicted in Fig. 1, G matches Q via strong simulation, in which the output node CL has two matches cl_{n-1} and cl_n, and $Q(G)$ is the set $\{cl_{n-1}, cl_n\}$.

Localized queries. A class of graph queries Q is said to have *data locality*, referred to as *localized queries*, if for any graph G and any node v in G, one can decide whether v is in $Q(G)$ locally, by inspecting only those nodes of G that are within d_Q hops of v, where d_Q is determined only by $|Q|$. Otherwise, the class of queries is called *non-localized*.

Both subgraph and simulation queries are localized. To compute $Q(G)$, we only need to visit those nodes within d_Q hops of v_p in G, where d_Q is the diameter of Q, $d_Q \leq |Q|$, and v_p is the match of the personalized node u_p of Q. That is, we only need to consider the d_Q-neighborhood $G_{d_Q}(v_p)$ of v_p in G. However, $G_{d_Q}(v_p)$ may be large [1].

Reachability queries. As an example of non-localized queries, we consider *reachability queries*. Given G and query Q as a pair of nodes (v_p, v_o) in G, it returns true if and only if v_p can reach v_o in G, *i.e.,* there is a path from v_p to v_o.

For instance, Example 1 gives an reachability query, to test whether Michael can reach Eric via social links.

Reachability queries are non-local: to compute $Q(G)$, we often have to visit nodes that reach v_p or v_o with a path of *unbound* length, even all the nodes in G in the worst case.

The notations of this paper are summarized in Table 1.

3. RESOURCE-BOUNDED QUERYING

To process a query Q on a big graph G while our resources are limited, we propose to identify and fetch a fraction G_Q of G within a given bound on its size, and compute approximate answers $Q(G_Q)$ with accuracy guarantees.

Accuracy of query answers. We define an accuracy measure for pattern queries, and then revise it for reachability.

Graph patterns. The exact answer to a pattern Q in G is a set $Q(G)$ of matches. Suppose that an algorithm $\mathcal{A}$ computes a set Y of approximate answers to Q in G. We define the *precision and recall of* Y *for* (Q, G) in the standard way:

$$\mathsf{precs}(Q, G, Y) = \frac{|Y \cap Q(G)|}{|Y|}, \ \mathsf{recall}(Q, G, Y) = \frac{|Y \cap Q(G)|}{|Q(G)|}.$$

That is, precs is the ratio of the number of correct matches in Y to the total number of matches in Y, while recall is the ratio of the number of correct matches in Y to the total number of matches in $Q(G)$. Based on these, we define the *accuracy of* Y *for* (Q, D) as the usual F-measure:

$$\mathsf{accuracy}(Q, G, Y) = 2\ \frac{\mathsf{precs}(Q, G, Y)\ \mathsf{recall}(Q, G, Y)}{\mathsf{precs}(Q, G, Y) + \mathsf{recall}(Q, G, Y)}.$$

The larger $\mathsf{accuracy}(Q, G, Y)$ is, the more accurate Y is.

When both $Q(G)$ and Y are $\emptyset$, *i.e.,* no match exists, we treat $\mathsf{accuracy}(Q, G, Y)$ as 1; we consider precs only if $Q(G)$ is $\emptyset$ but Y is not, and recall only if Y is $\emptyset$ but $Q(G)$ is not.

Reachability queries. When Q is a reachability query, $Q(G)$ is a single truth value. Given a set $\mathcal{Q}$ of reachability queries, we denote by $\mathcal{Q}(G)$ the set of exact answers for all queries Q in $\mathcal{Q}$, and by $\mathcal{Y}$ the set of truth values computed by algorithms $\mathcal{A}$ for $Q \in \mathcal{Q}$. Then we define $\mathsf{precs}(\mathcal{Q}, G, \mathcal{Y})$, $\mathsf{recall}(\mathcal{Q}, G, \mathcal{Y})$ and $\mathsf{accuracy}(\mathcal{Q}, G, \mathcal{Y})$ in the same way as above. Here $\mathsf{precs}(\mathcal{Q}, G, \mathcal{Y})$ is the ratio of the number of true positives and true negatives to the total number of answers returned by $\mathcal{A}$, which also include false positives and false negatives; similarly for $\mathsf{recall}(\mathcal{Q}, G, \mathcal{Y})$.

Resource-bounded query answering. We now present resource-bounded algorithms. Let $\alpha \in (0, 1)$ be a *resource ratio*, and $\mathcal{L}$ be a class of queries (subgraph or simulation).

Given a graph G and a query Q in $\mathcal{L}$, an algorithm $\mathcal{A}$ for $\mathcal{L}$ queries *with resource-bound* α does the following:

- fetches a fraction G_Q of G such that $|G_Q| \leq \alpha|G|$, by visiting at most $\alpha * c * |G|$ amount of data in G; and
- computes $Q(G_Q)$ as approximate answers,

where c is a coefficient such that $\alpha * c < 1$.

We say that $\mathcal{A}$ has *accuracy guarantee* η for $\mathcal{L}$ if for all graphs G and all queries $Q \in \mathcal{L}$, $\mathsf{accuracy}(Q, G, Q(G_Q)) \geq \eta$.

Note that the accuracy ratio η is in the range $(0, 1]$. When $\eta = 1$, algorithm $\mathcal{A}$ finds *exact answers* for all graphs G and queries Q *i.e.,* with 100% accuracy.

Similarly such algorithms are defined for reachability.

As illustrated in Fig. 2, algorithm $\mathcal{A}$ consists of two steps.

(1) *Dynamic reduction*. Given a query Q, it reduces a possibly big G to a small G_Q within the bound. In contrast to graph indexing, compression and summarization that build *the same structure for all queries* (see Section 1), dynamic reduction finds G_Q with only information needed for an *input query* Q, and hence, allows higher accuracy. One can use any techniques for dynamic reduction, including those for data synopses such as sampling and sketching, as long as the process *visits a bounded amount of data in* G. The reduction process may use some auxiliary information (*e.g.,* indices) collected by *offline* preprocessing that is conducted *once-for-all*, to help us answer *all queries* posed on G.

(2) *Approximate query answering*. Algorithm $\mathcal{A}$ computes $Q(G_Q)$ by accessing $\alpha|G|$ amount of data rather than the entire G. When $\alpha = 0.0015\%$ and $|G|$ is 1PB, *e.g.,* G_Q is of GB size and $\mathsf{accuracy}(Q, G, Q(G_Q))$ is high (see Section 6).

Example 2: Recall Q and G from Fig. 1. Set resource ratio $\alpha = 1.6\%$, and $c = 1$. Suppose that $m = 96$ and $n = 900$, *i.e.,* there are 1000 nodes within 2 hops of node *Michael*. Then a resource-bounded algorithm $\mathcal{A}$ is allowed to visit at most 16 nodes and edges in G. Ideally, $\mathcal{A}$ visits *Michael*, cc_1, cc_3, cl_{n-1}, cl_n and hg_m, and finds G_Q to be the subgraph induced by the nodes (with 14 nodes and edges). If so, $\mathcal{A}$ can find $Q(G_Q) = \{cl_{n-1}, cl_n\}$ and $\mathsf{accuracy}(Q, G, Q(G_Q)) = 100\%$. □

Remarks. The bound $\alpha|G|$ is essential to bounding *e.g.,* time, space and energy [6, 15, 18]. Disk-based algorithms

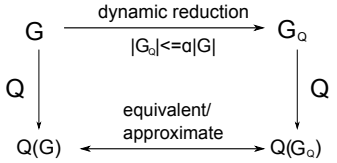


Figure 2: Resource-bounded query answering

(*e.g.,* [18]) can be used for dynamic reduction with disk reads for $O(\alpha * c * |G|)$ data, without loading the entire G. Note that the cost of *online* query processing involves only fetching G_Q by visiting $O(\alpha * c * |G|)$ data, and does not include the cost of offline preprocessing for auxiliary structures.

Fundamental problems and complexity. For a class $\mathcal{L}$ of graph queries, the problem of *resource-bounded query answering* is to find, given $\alpha \in (0, 1)$ and $\eta \in (0, 1]$, an algorithm with resource bound α and accuracy guarantee η.

This problem is hard. To see this, consider the following decision problem. The *exact resource-bounded querying* for $\mathcal{L}$ is to decide, given G and $Q \in \mathcal{L}$, whether there exists a subgraph G_Q of G such that $|G_Q| \leq \alpha|G|$ and $Q(G) = Q(G_Q)$. That is, whether there is a bounded G_Q at all that gives 100% accuracy for Q. The result below shows that this problem is already intractable for *localized* graph queries.

Theorem 1: *The exact resource-bounded querying problem is* NP*-hard for (a) simulation queries even when Q is a path and G is a DAG; and for (a) subgraph queries.* □

Proof sketch: (a) For simulation, we show that it is NP-hard by reduction from the set covering problem SCP, which is NP-complete (cf. [24]). Given an instance of SCP consisting of a finite set X, a family $F = \{C_1, \ldots, C_n\}$ of subsets of X and a positive integer k, we define a path pattern Q of length 2, a DAG G with elements of X and F as its nodes, and α as a function of $k, |X|, |F|$. We show that there is G_Q with $|G_Q| \leq \alpha|G|$ and $Q(G) = Q(G_Q)$ by strong simulation if and only if there exist k subsets in F whose union is X.
(b) For subgraph queries we use reduction from the subgraph isomorphism problem, which is NP-complete (cf. [24]). □

For reachability queries (non-localized), it is even worse. Here we consider algorithms that traverse a graph by following edges as usual, without precomputed indices.

Theorem 2: *For any $\alpha < 1$, there exists no algorithm for answering reachability queries that visits at most an α-fraction of G and has 100% accuracy guarantee.* □

Proof sketch: Assume by contradiction that such an algorithm $\mathcal{A}$ exists. We construct two graphs G_1 and G_2 such that the $\lceil \frac{1}{\alpha} \rceil$-neighborhoods of v_p and v_o in the two graphs are isomorphic to each other. To test whether v_p reaches v_o, we show that $\mathcal{A}$ returns true on both G_1 and G_2, while it should be true on G_1 and false on G_2. □

Not all is lost. In Sections 4 and 5, we develop resource-bounded algorithms for pattern and reachability queries, respectively, which often find 100% accurate answers even for very small α in real-life graphs (Section 6). That is, resource-bounded query answering is effective in practice.

4. ANSWERING LOCALIZED QUERIES

We now study resource-bounded algorithms for answering simulation and subgraph queries. This is nontrivial: Theorem 1 tells us that it is intractable to decide whether there exists a subgraph G_Q within a bound that preserves $Q(G)$.

Despite this, we develop resource-bounded algorithms for graph pattern queries that still have certain performance guarantees. The main result of the section is as follows.

Theorem 3: *There exist resource-bounded algorithms for simulation and subgraph queries such that given any resource ratio $\alpha \in (0, 1)$, graph G and query Q,*

(a) they find a subgraph G_Q of G with $|G_Q| \leq \alpha|G|$, by visiting at most $d_G * \alpha|G|$ nodes and edges in G, in $O(d_G|Q||G_Q|)$ time; and

(b) $Q(G_Q)$ has 100% accuracy when $\alpha \geq \frac{2((l*f)^d - 1)}{(l*f-1)|G|}$. □

Here d_G is the maximum degree of nodes in $G_{d_Q}(v_p)$ (corresponding to parameter c in resource-bounded query answering), d is the diameter of Q when Q is treated as an undirected graph, l is the number of distinct labels in Q, and f is the maximum number of the nodes in $G_{d_Q}(v_p)$ that have the same label and a common parent or child.

Theorem 3 tells us that we can effectively find small G_Q by accessing a bounded amount of data in G. Moreover, for small α, we have 100% accuracy. Indeed, in practice d_G, l, d and f are all quite small: (1) on average d_G is around 190 in Facebook [34]; this bound also applies to f; (2) d and l are smaller than $|Q|$, and $|Q|$ is small in personalized social search [8] and ego network analysis [22]. In our experimental study using real-life graphs, while $|G_{d_Q}(v_p)|$ is up to 0.01% of $|G|$ with d_G up to 483, we find that we consistently get 100% accuracy even when α is 0.0015%, which is on average 3% of the theoretical bound given in Theorem 3(b), where $|G_Q|$ is up to 19% of the size $|G_{d_Q}(v_p)|$ (see Section 6).

We next prove Theorem 3 for simulation queries first (Section 4.1), and then adapt it to subgraph queries (Section 4.2). We focus on dynamic reduction to find G_Q; after that, we simply use *existing algorithms* for strong simulation [20] and subgraph isomorphism [11] to compute $Q(G_Q)$.

4.1 Resource-Bounded Strong Simulation

We start with a resource-bounded algorithm for simulation queries, denoted by RBSim. Given a simulation query Q, a graph G and a resource ratio α, RBSim finds a subgraph G_Q of $G_{d_Q}(v_p)$ with $|G_Q| \leq \alpha|G|$, by visiting a $d_G * \alpha|G|$-fraction of G. It returns $Q(G_Q)$ as approximate answers.

The tricky part of RBSim is its dynamic reduction strategy to induce subgraph G_Q. One might want to take G_Q as d_Q-neighborhood $G_{d_Q}(v_p)$ and compute $Q(G_Q)$. However, $G_{d_Q}(v_p)$ easily exceeds resource bound. Continuing with Example 2, the 2-neighbor of *Michael* has 1000 nodes, exceeding the bound when $\alpha = 1.6\%$. To cope with this, RBSim performs a *controlled traversal* of G starting from the match v_p of the personalized node u_p, and populates G_Q as follows. (a) Its search is guided by Q, and includes in G_Q only *candidate matches* of query nodes. (b) It maintains dynamically updated weights for nodes v of G, indicating how likely v can contribute to $Q(G)$. It only adds to G_Q those nodes with top-ranked weights until G_Q reaches the bound $\alpha|G|$. (c) It uses a dynamically maintained bound to control the number of candidate in G_Q for each query node u. This ensures that each u has a fair chance of finding a match in G_Q and avoids bias towards high-degree nodes.

Below we first introduce our node-selection strategy for G_Q. We then give the details of RBSim and its analyses.

Dynamic reduction. To populate G_Q, for each node v, RBSim maintains (a) the degree $d(v)$ of v, *i.e.,* the cardinal-

ity of its 1-neighborhood $N_1(v)$ (or simply $N(v)$), consisting of the parents and children of v; and (b) a set $\mathcal{S}_l$ of pairs (ℓ, g), where ℓ is a distinct label from $N(v)$, and g is the number of occurrences of ℓ in $N(v)$. These can be found by a linear traversal of G in an once-for-all *offline preprocessing*.

Example 3: Consider graph G of Fig. 1. Let $|G| = 1000$, where $m = 96$ and $n = 900$. An offline preprocessing step computes, for node *Michael*, (a) $\mathcal{S}_l = \{$(HG, 96), (CC, 3)$\}$, *i.e.,* there are 96 HG nodes and 3 CC nodes in the neighbors of *Michael*, and (b) 99 as its degree. Similarly, for hg_m, $\mathcal{S}_l$ = $\{$(*Michael*, 1), (CL, 3)$\}$ and its degree = 4. □

For a node v in G and a query node u in Q, to decide whether to include v in G_Q as a candidate match of u, we consider the weight of v defined in terms of the following.

(1) A Boolean *guarded condition* $C(v, u)$ indicating whether v is a candidate match of u. We define $C(v, u) =$ true if and only if $f_v(u) = L(v)$, and for each parent (resp. child) u' of u in Q, there exists a parent (resp. child) v' of v in $N(v)$ with $f_v(u') = L(v')$. We use $C(v, u)$ to filter nodes that are not matches, and hence reduce the search space. Indeed, if $C(v, u)$ is false, then v is not a match of u by strong simulation (Section 2). Using the auxiliary structure $\mathcal{S}_l$ and hashing function, $C(v, u)$ can be evaluated efficiently.

(2) A dynamically maintained cost $c(v, u)$. It is the total number of nodes u' of $N(u)$ in Q that do not find v' of $N(v)$ in G_Q such that $C(u', v') =$ true in G_Q. Intuitively, $c(v, u)$ indicates if v is added to G_Q, the number of additional nodes in $N(v)$ that may also be included in G_Q so that v can match u. The larger $c(v, u)$ is, the more costly v is for G_Q.

(3) A dynamically maintained value $p(v, u)$, indicating the probability for v to match u in G_Q. It is the total number of nodes v' in $N(v)$ that satisfy $C(u', v') =$ true, for all $u' \in N(u)$, which are candidates for u' if added to G_Q. Note that $p(v, u)$ can be extended by incorporating statistics from query log, such as the "activeness" of a user, user search interests [7], or topological importance such as centrality.

(4) A dynamically adjusted bound b such that at most $\min(b, p(v, u))$ nodes in $N(v)$ are visited for a query node u if v is to be added to G_Q. We use b to reduce the chance of populating G_Q with too many nodes from "dense" regions of G, when, *e.g.,* v has a large number of candidate matches in $N(v)$. In this way, each v has a more "equal" chance to be explored. This can be extended by making use of sampling.

Based on these, our node selection strategy is as follows. Suppose that we are at node v_1 and want to pick a node v in $N(v_1)$ to include in G_Q as a candidate match of u. We select v if (a) $C(v, u)$ is true, and (b) *the estimated weight* $\frac{p(v,u)}{c(v,u)+1}$ is the maximum among all those in $N(v_1)$. That is, we favor nodes with high potential and low estimated cost.

Algorithm. We are now ready to give algorithm RBSim, shown in Fig. 3. Its main driver is simple: it first calls procedure Search to fetch a subgraph G_Q (line 1). It then computes $Q(G_Q)$ with the algorithm of [20] (line 2), and returns the set of the matches of the output node u_o in G_Q (line 3).

Procedure Search. Given Q, G and α, the procedure identifies a subgraph G_Q with $|G_Q| \leq \alpha|G|$, as shown in Fig. 3. It starts with an empty G_Q (line 1), and initializes a stack S with the pair (u_p, v_p) of the personalized node and its match (line 2). It then traverses G starting from v_p, and populates

Algorithm RBSim

Input: A query Q, a graph G, a resource ratio α.
Output: Approximate answers $Q(G_Q)$.

1. $G_Q :=$ Search(Q, G, α);
2. $Q(G) :=$ Match(Q, G_Q);
3. **return** $Q(G)$;

Procedure Search

Input: Q, G and α.
Output: Subgraph G_Q.

1. initialize graph $G_Q := \emptyset$; $b := 2$;
2. Stack S.push (u_p, v_p); terminate:=false; changed:= false;
3. **while** terminate $\neq$ true **do**
4. pair $(u, v) :=$ S.pop();
5. add v to G_Q if it is not already in G_Q;
6. update terminate; changed:= true if v is new to G_Q;
7. **if** terminate **then return** G_Q;
8. **for each** unvisited edge (u, u') or (u', u) **do**
9. ranked list $S_p :=$ Pick $(u', v, Q, G, G_Q,$ S$)$;
10. **for each** $v' \in S_p$ **do** S.push (u', v');
11. **if** changed and S $= \emptyset$ **then**
12. $b := b + 1$; S.push (u_p, v_p); changed:= false;
13. **if** not changed and S $= \emptyset$ **then** terminate := true;
14. **return** G_Q;

Figure 3: Algorithm RBSim

G_Q by including new nodes and edges, which are descendants or ancestors of v_p in its d_Q-neighborhood $G_{d_Q}(v_p)$ (lines 3–13). It uses two flags to control the traversal (line 2): (a) terminate becomes true if either $|G_Q| = \alpha|G|$, or no nodes within d_Q hops of v_p can be added to G_Q; and (b) changed is true if for a given selection bound b, there are new nodes added to G_Q, *i.e.,* there are still candidates within d_Q hops of v_p. The bound b is initially set 2 (line 1).

More specifically, the traversal is *guided by pattern* Q. If a new node v is added to G_Q as a candidate for query node u (line 5, initially v_p), we set changed true (line 6). If now G_Q reaches the bound $\alpha|G|$, then G_Q is returned (line 7). Otherwise, it inspects both *children and parents* u' of query node u (line 8). For each such u', it calls procedure Pick (line 9) to select a ranked list S_p of best new candidates v' for u', from the neighborhood $N(v)$ of v in G, where $|S_p|$ is *bounded* by b. Each pair (u', v') is then pushed onto the stack S, with the best candidate v' at the top of S (line 10).

If at this stage, stack S is empty (*i.e.,* no new insertions) but changed is true (*i.e.,* there are still match candidates that are not yet in G_Q), we increase b and start the search from (u_p, v_p) again to find them (line 12). If S is empty and changed is false, no more nodes can be added to G_Q, and we set terminate true (line 13). The process proceeds until terminate becomes true, and then G_Q is returned (line 14).

Procedure Pick. Given a node u' in Q and a node v in G, Pick (omitted) finds a list S_p of "top-ranked" nodes v' in $N(v)$ that are not yet in the stack S. To do this, Pick keeps a max-heap to store $N(v)$, with the estimated weight as the sorting key. For nodes v' in the max-heap that are not added to G_Q yet (indicated by a dynamically maintained Boolean flag), it first checks whether the guarded condition $C(v', u')$ = true, and then updates $c(v', u')$ and $p(v', u')$ by checking the neighborhoods $N(v')$ in G_Q and $N(u')$ in Q. It returns S_p with the top-b ones with the maximum weights in the max-heap that satisfy the guarded condition, where b is the selection bound. Note that S_p is possibly empty if no new candidates exist (*e.g.,* all nodes in $N(v)$ are added to G_Q).

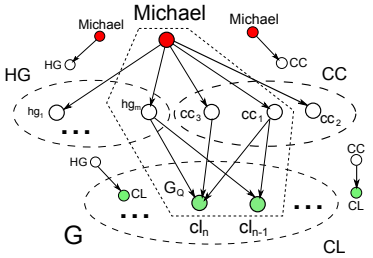


Figure 4: Resource-bounded simulation

Example 4: Consider Q and G of Fig. 1. When $\alpha = 1.6\%$ (and $c = 1$), procedure Search finds a subgraph G_Q of G (Fig. 4) with size no more than 14, and visits no more than 16 nodes and edges as follows. (1) It first pushes the pair (Michael, *Michael*) onto stack S (line 2), and adds *Michael* to G_Q (line 5). (2) It then checks edge (Michael, CC) in Q, and invokes procedure Pick to find top 2 candidates for CC. In this case, Pick returns cc_1 and cc_3 (to be explained soon). Hence, Search pushes (CC, cc_1) and (CC, cc_3) onto S (line 10), and inserts cc_3, the current top of S, into G_Q (line 5). (3) In the same way, it processes query edge (CC, CL), and then a "backward" edge (HG, CL). It includes in G_Q three new nodes cl_n, cl_{n-1} and hg_m, along with edges between them. It next traces back to query edge (Michael, HG). As the parent of hg_m is already in G_Q, no new node is added. (4) Moving up the stack S, Search backtracks to cl_{n-1} and cc_1. As their neighborhoods are all in G_Q, Search finally pops up the first pair (Michael, *Michael*), which makes S empty. At this moment G_Q already reaches its size bound 14. Hence, Search sets terminate true, and returns G_Q. RBSim then invokes Match [20] to compute two matches cl_n and cl_{n-1} from G_Q, for the output node CL in Q. In the entire process, 16 nodes and edges are visited. Note that RBSim visits each query edge once (line 8).

We now show how procedure Pick works. When Pick is invoked by Search for edge (Michael, CC), Pick rules out the node cc_2 since it does not satisfy the guarded condition, *i.e.,* it has no *CL* child as required by the query node CC. For the two remaining nodes cc_1 and cc_3, it looks up the auxiliary structure $\mathcal{S}_l$, and finds that (a) both have a cost 1, since query node CC requires a *CL* child of them to be in G_Q, and its parent Michael already has a candidate *Michael* in the current G_Q; and (b) $p(cc_1, \text{CC}) = 3$, indicating that there are 3 possible matches in $N(cc_1)$, while $p(cc_3, \text{CC}) = 2$. Pick returns list $S_p = [cc_1, cc_3]$ when the bound $b = 2$.

For edge (HG, CL) in Q, Pick traces back to hg_m, a parent of cl_n and cl_{n-1}. It updates the cost of hg_m from 1 to 0, as it already has a child cl_n and parent *Michael* in G_Q, while $p(hg_m, \text{HG}) = 4$. Pick finds node hg_m from the max-heap. Note that all the other HG nodes have cost 1, but do not get into G_Q as they have no CL child. □

Performance analysis. We now prove Theorem 3 by analyzing algorithm RBSim. (1) RBSim extracts a subgraph G_Q with $|G_Q| \leq \alpha|G|$, guaranteed by the termination condition. (2) For each newly added node v to G_Q, procedure Search inspects at most 1-hop of v in G, by calling Pick. Hence Search visits at most $d_G * \alpha|G|$ nodes or edges, where d_G is the maximum node degree in $G_{d_Q}(v_p)$. (3) For time complexity, note that Search executes the **while** loop (lines 3-13) at most $\alpha|G|$ times. This is because (a) at least one new node is added to G_Q in each loop, and (b) $|G_Q| \leq \alpha|G|$. For each node v and query node u', it checks the guarded conditions in $O(d_G)$ time, and maintains the max-heap in $\log |d_G|$ time. As there are in total $|V_Q|$ query nodes, it takes at most $O(d_G|Q||G_Q|)$ time. These verify Theorem 3(a).

We next prove Theorem 3(b) by induction on the diameter d of Q (when G is treated as an undirected graph). The case when $d = 1$ is trivial. When $d = 2$, RBSim finds G_Q, in the worst case, a two-level "tree" rooted at v_p with size at most $1 + 2 * l * f \leq 2(\frac{(l*f)^2-1}{l*f-1})$. Now assume Theorem 3(b) holds when $d = k$. That is, RBSim identifies G_Q as a k-level, $l * f$-ary "tree", which contains all possible matches for Q in G. For $d = k + 1$, RBSim only needs to explore at most $l * f$ children for each leaf in G_Q to include any new matches at level $k+1$. The new G_Q hence has size at most $2\frac{(l*f)^{(k+1)}-1}{l*f-1}$. Hence, when $\alpha \geq \frac{2((l*f)^d-1)}{(l*f-1)|G|}$, $Q(G_Q) = Q(G)$, *i.e.,* RBSim finds G_Q of size $\alpha|G|$ and guarantees 100% accuracy.

Putting these together, Theorem 3 follows.

4.2 Resource-Bounded Subgraph Queries

We now outline a resource-bounded algorithm for subgraph queries, denoted by RBSub. It revises RBSim as follows: (1) we enrich the guarded condition and cost estimation for isomorphism test, and (2) after G_Q is found, we use a subgraph isomorphism algorithm [11] to compute $Q(G_Q)$.

More specifically, we use the same termination condition as in RBSim, but revise the guarded condition $C(v, u)$ for RBSub as follows: $C(v, u)$ is true if and only if for every query node $u' \in N(u)$ in Q with degree $d_{u'}$, there exists a distinct node $v' \in N(v)$ in G with the same label and degree $d_{v'} \geq d_{u'}$. That is, $C(v, u)$ imposes additional *degree constraints* for subgraph isomorphism. Accordingly, (a) for a node v in G and a node u in Q, RBSub defines estimated costs $c(v, u)$ and potential $p(v, u)$ by using the revised guarded condition $C(v, u)$; and (b) procedure Pick in RBSub favors candidates for query nodes with larger degree and lower costs.

One can verify Theorem 3 for subgraph queries along the same lines as the proof above for simulation queries.

5. NON-LOCALIZED QUERYING

We next study resource-bounded query answering for reachability. Despite of Theorem 2, we develop a resource-bounded algorithm that guarantees 100% true positives.

Theorem 4: *There exists a resource-bounded algorithm such that given any resource ratio $\alpha \in (0, 1)$, data graph G and reachability query Q, it*

(a) visits at most $\alpha|G|$ amount of data, by using an index of size $\alpha|G|$;
(b) takes $O(\alpha|G|)$ time to approximately answer $Q(G)$;
(c) and it returns true *only if $Q(G)$ is* true. □

The algorithm visits at most $\alpha|G|$ nodes and edges (hence parameter $c = 1$). It never returns "false positive". We find from our experimental study that the algorithm constantly achieves 100% accuracy even when $\alpha = 0.05\%$. Moreover, its precs and recall get higher over larger and denser G.

We give a constructive proof for Theorem 4 by providing the algorithm. It requires an *once-for-all* preprocessing that compress G and constructs a hierarchical indexing.

Preprocessing. An *once-for-all preprocessing* first reduces a (possibly cyclic) G to a directed acyclic graph (DAG) G_{DAG}, by using the compression method of [12]. It is reachability preserving, *i.e., for all reachability queries Q* posed on G, $Q(G) = Q(G_{\text{DAG}})$. The reason for this step is twofold: (a)

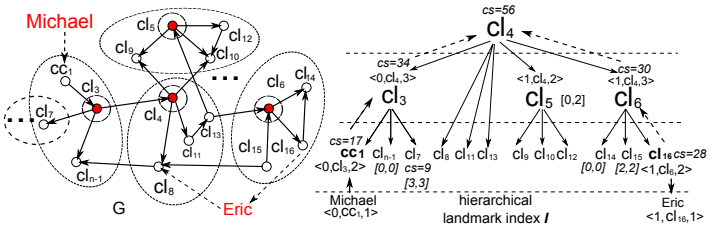


Figure 5: A hierarchical landmark index

G_{DAG} is much smaller than G while retaining reachability; and (b) hierarchical indexing (to be given below) is more effective on DAGs. Below we simply refer to the DAG as G.

Hierarchical indexing. An *hierarchical index* $\mathcal{I}$ is then constructed using landmarks [13]. Given a pair of nodes (v_1, v_2) in G such that v_1 reaches v_2, we say that a node v in G is a *landmark* for (v_1, v_2) and *covers* (v_1, v_2) if it is on a path from v_1 to v_2. We create $\mathcal{I}$ of size $\alpha|G|$, again *once for all* Q *on* G.

Query answering. Given a query $Q = (v_p, v_o)$ and $\mathcal{I}$, we check whether v_p reaches v_o by searching $\mathcal{I}$ instead of G, to find whether there is a landmark in $\mathcal{I}$ that covers (v_p, v_o).

Below we focus on hierarchical index (Section 5.1), and the resource-bounded reachability algorithm (Section 5.2).

5.1 Hierarchical Landmark Index

One might want to find a minimum set L_m of landmarks and take the subgraph induced by L_m as G_Q, such that every pair of connected nodes in G is covered by a landmark in G_Q. Given a query (v_p, v_o), we could then test whether v_1 reaches v_2 by checking whether there is a landmark in G_Q covering (v_p, v_o). However, such a G_Q may *exceed* $\alpha|G|$; moreover, the problem of finding a minimum L_m is intractable [13].

In light of this, given G and α, we build a *hierarchical landmark index* $\mathcal{I}$ of size $\alpha|G|$ to cover as many connected node pairs in G as possible. Below we use a to denote $\lfloor \frac{2}{\alpha} \rfloor$.

Index structure. The index $\mathcal{I}$ is a set of rooted trees (a forest), consisting of $\frac{\alpha|G|}{2}$ landmarks of G in total. Each tree has a depth of at most $\lfloor \log_a |G| + 1 \rfloor$, *i.e.,* it has at most $\lfloor \log_a |G| + 1 \rfloor$ levels with all its leaves at level 1. More specifically, (1) each node in $\mathcal{I}$ is a landmark of G, and (2) there is an edge (v_1, v_2) in $\mathcal{I}$ if and only if either v_1 can reach v_2 or v_2 can reach v_1. Intuitively, $\mathcal{I}$ organizes a set of landmarks into various "levels": (a) the leaves cover connected node pairs in G, and (b) those at level $i > 1$ specify the reachability among the landmarks at the lower levels.

Auxiliary information. We also maintain the following for reachability checking. (1) For each landmark v in $\mathcal{I}$, we use $v.\mathsf{cs}$ to store its *cover size*, *i.e.,* how many connected node pairs in G are covered by v. (2) For each edge (v_1, v_2) in $\mathcal{I}$, we define a label $v_2.e$: it is $< 1, v_1, i+1 >$ (resp. $< 0, v_1, i+1 >$) if v_1 can reach v_2 (resp. v_2 can reach v_1), where v_1 is at level $i+1$ and v_2 at level i. (3) For each node v that is in G but not in $\mathcal{I}$, we define a set $v.\mathcal{E}$ of triples such that for each leaf v' in $\mathcal{I}$, $< 1, v', 1 >$ (resp. $< 0, v', 1 >$) is in $v.\mathcal{E}$ if v can reach v' (resp. v' can reach v) by following a path that contains no landmark in $\mathcal{I}$. Note that $|v.\mathcal{E}| \leq \frac{\alpha|G|}{2}$. (4) We also define $v.d$ (resp. $v.r$), the degree (resp. the topological rank) of v in G (recall that G is a DAG). Here $v.r$ is defined as follows: (a) $v.r = 0$ if v has no child in G; (b) otherwise, $v.r = v'.r+1$, where v' is the child of v with the largest rank. (4) For each landmark v in $\mathcal{I}$, we define its *topological range* $v.R = [r_1, r_2]$, where r_1 (resp. r_2) is the smallest (resp. largest) rank of the landmarks in the subtree rooted at v in $\mathcal{I}$. The range is simply $[v.r, v.r]$ if v is a leaf in $\mathcal{I}$.

Procedure RBIndex
Input: A graph $G=(V, E, L)$ and α.
Output: A hierarchical landmark index $\mathcal{I}$.

1. $\mathcal{I}$:=$\emptyset$;
2. greedily select $\lfloor \frac{\alpha|G|}{2} \rfloor$ landmarks from V as LM_1;
3. construct landmark graph G_1 from LM_1 and G;
4. update $\mathcal{I}$ with LM_1;
5. **for** l from 2 to $\lfloor \log_a |G| \rfloor + 1$ **do** /* $a = \lfloor \frac{2}{\alpha} \rfloor$ */
6. greedily select $\lfloor \frac{\alpha|G_{l-1}|}{2} \rfloor$ landmarks LM_l from G_{l-1};
7. expand $\mathcal{I}$ with LM_l; /*move nodes in LM_l up and add edges*/
8. Encode ($\mathsf{LM}_l, \mathsf{LM}_{l-1}, G_{l-1}$);
9. construct landmark graph G_l from LM_l and G_{l-1};
10. **for each** $v \in V \setminus \mathsf{LM}_1$ **do**
11. assign labels $v.\mathcal{E}$ in terms of the leaves of $\mathcal{I}$;
12. **return** $\mathcal{I}$;

Figure 6: Procedure RBIndex

One may verify that $\mathcal{I}$ guarantees the following.

Lemma 5: *For any nodes* v *and* v' *in* G, *(1)* v *can reach* v' *if there exist landmarks* v_1, v_2, v_3 *in* $\mathcal{I}$ *such that* v *reaches* v_1, v_2 *reaches* v', $v_1.e =< 1, v_3, i >$ *and* $v_2.e =< 0, v_3, i >$; *(2) for any* v_4 *in* $\mathcal{I}$ *with* $v_4.R = [r_1, r_2]$, *if* $r_2 \leq v'.r$ *or* $r_1 \geq v.r$, *then no node in the subtree rooted at* v_4 *covers* (v, v'). □

Example 5: Consider a DAG G shown in Fig. 5, with $|G| = 128$. Let $\alpha = 0.25$ and $c = 1$. We show an index $\mathcal{I}$ with 16 nodes (each corresponds to a landmark in G) and 15 edges in Fig. 5. Observe the following. (1) Cover size $cl_4.\mathsf{cs} = 56$, *i.e., cl*$_4$ covers 56 connected node pairs in G. (2) Edge (cl_3, cc_1) in $\mathcal{I}$ with the label $< 0, cl_3, 2 >$ of cc_1 indicates that cc_1 reaches cl_3. (3) The label set of Michael (which is not in $\mathcal{I}$) is {<0,cc_1,1>}, which indiates that Michael can reach cc_1 in $\mathcal{I}$ by only accessing the nodes "outside" $\mathcal{I}$. One may further verify that Michael reaches Eric in G by using $\mathcal{I}$. □

Algorithm. We now present an algorithm, denoted by RBIndex (Fig. 6), that builds a hierarchical index $\mathcal{I}$ of size $\alpha|G|$. It selects a set LM_1 of $\lfloor \frac{\alpha|G|}{2} \rfloor$ landmarks of G (line 2), constructs a *landmark graph* G_1 such that LM_1 is its node set, and there exists an edge (v_1, v_2) in G_1 if and only if v_1 can reach v_2 in G (line 3). The nodes of LM_1 are added to (initially empty) $\mathcal{I}$ as its level-1 nodes (leaves; line 4).

RBIndex then expands $\mathcal{I}$ "bottom-up" (lines 5-9). For each $l \in [2, \lfloor \log_a |G| + 1 \rfloor]$, it does the following. (1) It selects a set LM_l of $\lfloor \frac{\alpha|G_{l-1}|}{2} \rfloor$ landmarks from G_{l-1} following a greedy strategy (to be described later; line 6). (2) It then moves the landmarks in LM_l *up one level* in $\mathcal{I}$ to be its nodes at level $l+1$. For each landmark v in LM_l, it adds an edge from v to a node v' at level $l-1$ of $\mathcal{I}$ if v can reach v' or can be reached from v' in G_{l-1} (line 7). Accordingly, it assigns a label to v', and updates the topological range of v (line 8), by invoking procedure Encode (omitted). We then build the landmark graph G_l from LM_l and G_{l-1} (line 9), such that LM_l is the node set of G_l, and G_l has an edge (v_1, v_2) if and only if v_1 can reach v_2 in G_{l-1}.

The process above repeats until only a single landmark remains (*i.e.,* at level $\log_a |G|$; line 5). RBIndex then assigns labels to the rest of nodes in G as described earlier (lines 10-11). After this, it returns the index $\mathcal{I}$ (line 12).

Landmark selection. For each G_l, we want to select a set LM_l of $\frac{\alpha|G_{l-1}|}{2}$ landmarks to cover a maximum number of node

Procedure RBReach

Input: A reachability query $Q = (v_p, v_o)$, and index $\mathcal{I}$ of size $\alpha|G|$.
Output: Approximate answers to $Q(G)$.

1. terminate:= false; answer:=false;
2. active set v_p.Active := $\{v|\ (1, v, l) \in v_p.\mathcal{E}\ \} \cup \{v_p\}$;
3. active set v_o.Active := $\{v|\ (0, v, l) \in v_o.\mathcal{E}\ \} \cup \{v_o\}$;
4. update terminate and answer;
5. **if** answer = true **then return** answer;
6. **while** terminate $\neq$ true **do**
7. v_p.Active := PickLM (v_p.Active, $\mathcal{I}$);
8. v_o.Active := PickLM (v_o.Active, $\mathcal{I}$);
9. update terminate and answer;
10. **if** answer = true **then return** answer;
11. **if** no node can be added to v_p.Active and v_o.Active **then**
12. terminate:=true;
13. **return** answer;

Figure 7: Procedure RBReach

pairs. Unfortunately, this problem is NP-hard [13]. In light of this, we use a greedy strategy based on the topological ranks and degrees of landmarks. (1) We select a landmark v with the maximum $\frac{v_d * v_r}{L * D}$, where L and D are the maximum topological rank and node degree in G_l, respectively. Intuitively, the higher $v.r$ and $v.d$ are, the more likely v covers more connected node pairs. (2) We then remove from G_l node v and a nodes that connect to v (if they exist), where $a = \lfloor \frac{2}{\alpha} \rfloor$, and select the next best node from the updated G_l. We repeat (1) and (2) until we find $\frac{\alpha|G_{l-1}|}{2}$ landmarks.

Example 6: Given G of Fig. 5 and $\alpha = 0.25$, RBIndex builds the index $\mathcal{I}$ of Fig. 5 as follows. (1) It first greedily selects 16 landmarks from G and constructs landmark graph G_1 with them. (2) It then select 4 landmarks cl_3, cl_4, cl_5 and cl_6 from G_1, as nodes at level 2 or higher in $\mathcal{I}$. Edges are added from these nodes to those at level 1, *e.g.,* (cl_3, cc_1), as well as cover sizes, *e.g.,* cl_4.cs = 56. New labels (resp. topological ranges) are assigned to nodes at level 1 (resp. level 2), *e.g.,* $< 0, cl_1, 2 >$ (resp. $[0, 2]$ for cl_5). It then builds G_2 from these landmarks and G_1. Finally, RBIndex selects cl_4, and moves it up one level from level 2 in $\mathcal{I}$ as its third level landmark. It adds edges from cl_4 to level-2 landmarks, adjusts labels, cover sizes and ranges. □

Analysis. One can verify that $\mathcal{I}$ contains at most $\frac{\alpha|G|}{2}$ landmarks and hence, $\alpha|G|$-1 nodes and edges as a forest. It has at most $\lfloor \log_a |G| \rfloor$+1 levels, where $a = \lfloor \frac{2}{\alpha} \rfloor$. For a graph of size 1PB (10^{15}), when α is 0.2%, $\mathcal{I}$ has at most 6 levels. Algorithm RBIndex takes at most $O(|G| + (\alpha|G|)^2)$ time: (a) it takes $O(|G|)$ time to select $\frac{\alpha|G|}{2}$ landmarks, and (b) it takes $O((\alpha|G|)^2)$ time ito test reachability between the landmarks and the other nodes in G.

5.2 Resource-Bounded Reachability

We next present a resource-bounded algorithm for checking reachability after the compression of G and the creation of index $\mathcal{I}$. It is denoted as RBReach and shown in Fig. 7. Procedure RBReach performs a "bi-directional" search on the index $\mathcal{I}$, starting from landmarks in $v_p.\mathcal{E}$ and $v_o.\mathcal{E}$. At each landmark v in $\mathcal{I}$, RBReach checks whether the condition of Lemma 5(1) is satisfied by the landmarks visited so far, and returns true if so. Otherwise it either "rolls-up" to its parent in $\mathcal{I}$, or "drills-down" to its children, to inspect more landmarks. It returns false if all the landmarks have been visited but the condition of Lemma 5(1) is still not met.

Drill down or roll up. To decide whether to roll up or drill down at a landmark v of $\mathcal{I}$, RBReach dynamically maintains the following. (1) *Boolean guarded condition* $C(v, v_p, v_o)$, indicating whether v can possibly reach v_o via v_p. We define $C(v, v_p, v_o)$ = true if and only if for the topological range $v.R = [r_1, r_2]$, $r_2 > v_o.r$ and $r_1 < v_p.r$. We filter the entire subtree rooted at v if $C(v, v_p, v_o)$ = false (see Lemma 5(2)). (2) *Cost* $c(v)$, defined as the size of the subtree rooted at v in $\mathcal{I}$, excluding the total size of the subtrees rooted at its children that are already visited in $\mathcal{I}$. The larger $c(v)$ is, the more landmarks need to be inspected. (3) *Potential* $p(v)$, which is the cover size v.cs subtracted by the sum of the cover sizes of its children that have been visited. The higher $p(v)$ is, the more likely that v connects to v_p or v_o. We define *the weight* $w(v)$ of v to be $\frac{p(v)}{c(v)+1}$ if $C(v, v_p, v_o)$ = true, and $w(v) = -\infty$ otherwise. At landmark v of $\mathcal{I}$, we roll up to its parent v' if $w(v')$ is the maximum, and drill down to a child v'' if $w(v'')$ is the largest, if the edge $(v'v)$ or (v, v'') is not already visited, respectively.

Algorithm. We now present RBReach. It uses two Boolean flags to control the search: answer is true if it finds that v_p reaches v_o, and terminate is true if all the landmarks in $\mathcal{I}$ have been visited. Initially, both are false (line 1). RBReach keeps track of the landmarks that v_p can reach and those that can reach v_o, in sets v_p.Active and v_o.Active, respectively, initially extracted from $v_p.\mathcal{E}$ and $v_o.\mathcal{E}$ (lines 2 and 3). To update terminate, it also maintains a set consisting of landmarks that are already visited (not shown).

After v_p.Active and v_o.Active are initialized, RBReach checks whether v_p can already be decided to reach v_o; if so, it returns true (lines 3-4). Otherwise, it iteratively expands v_p.Active and v_o.Active by including landmarks in $\mathcal{I}$ that are reachable from the nodes in v_p.Active (line 7) and that can reach the nodes in v_o.Active (line 8), respectively. This is done by procedure PickLM (not shown), which rolls up or drills down $\mathcal{I}$ following the strategy described above. When new landmarks are added, termination and answer are updated accordingly (line 9). We set answer true if there exists a landmark in both v_p.Active and v_o.Active, *i.e.,* the condition of Lemma 5(1) is satisfied. If so, it returns true (line 10). If the set visited includes all the nodes in $\mathcal{I}$, termination is set true (lines 11-12), and false is returned (line 13). To efficiently decide whether v_p.Active and v_o.Active share a node, RBReach stores a flag with a value "v_p" or "v_o" at each node to indicate if it is already in v_p.Active or v_o.Active. iThis allows us to check Lemma 5(1) with little extra time.

Example 7: Given the index $\mathcal{I}$ of Fig. 5, RBReach checks whether Michael can reach Eric as follows. (1) It starts with Michael.Active = $\{cc_1\}$ and Eric.Active = $\{cc_{16}\}$. (2) As termination and answer are false, RBReach calls PickLM, which rolls up to cl_3 from cc_1. We add cl_3 to Michael.Active. Similarly, cl_6 is added to Eric.Active. (3) PickLM finds that cl_4 has weight $w(cl_4) = \frac{46}{9} = 5.1$ (after visiting cl_3 and cl_6, the cost $c(cl_4)$ is now $16-8 = 8$, and its potential is updated to $56 - 10 = 46$, with $p(cl_3) = 34 - 26 = 8$, and $p(cl_6) = 30-28 = 2$). In contrast, $w(cl_7) = \frac{9}{2} = 4.5$, and the guarded condition of cl_{n-1} is false since Eric has a topological rank 2 but the range of cl_{n-1} is $[0, 0]$. Hence, it decides to roll up to cl_4 from cl_3 rather than to drill down. (4) For the same reason, it rolls up to cl_4 from cl_6. Now cl_6 is in both Michael.Active and Eric.Active, and true is returned. □

Analysis. To show Theorem 4, observe the following. (1) RBReach visits at most $\alpha|G|$ amount of data. In the worst case, it visits the entire $\mathcal{I}$. As shown in Section 5.1, $|\mathcal{I}| \leq \alpha|G| - 1$. (2) It answers Q in $O(\alpha|G|)$ time, since it visits each edge in $\mathcal{I}$ at most twice. Moreover, RBReach only needs to check the flag of each newly added landmark to test the condition of Lemma 5(1), as remarked earlier. As the edge number is no larger than $\frac{\alpha|G|}{2} - 1$, the total time is hence in $O(\alpha|G|)$. (3) RBReach returns true only when there exists a landmark v in both v_p.Active and v_o.Active. By Lemma 5(1), $Q(G)$ is true. Hence, it guarantees 100% true positives.

6. EXPERIMENTAL STUDY

Using real-life and synthetic data, we conducted two sets of experiments to evaluate the accuracy, efficiency and scalability of our resource-bounded algorithms.

Experimental setting. We used two real-life datasets: (a) *Youtube*[3], a video sharing network with 1,609,969 nodes (videos) and 4,509,826 edges (recommendations); and (b) *Yahoo*[4], a snapshot of Yahoo Web graph with 3,000,022 nodes (Web pages) and 14,979,447 edges (links). We also designed a generator to produce synthetic graphs $G = (V, E, L)$, controlled by the numbers of nodes $|V|$ and edges $|E|$, for L from a set Σ of 15 labels.

Query generator. We generated patterns controlled by the number $|V_p|$ of query nodes and the number $|E_p|$ of query edges. For patterns on real-life graphs, their labels were drawn from those datasets, and for synthetic graphs, they came from the alphabet Σ. We randomly selected a personalized node and an output node for each query. For reachability tests, we randomly sampled a set of ordered node pairs from a data graph, each pair representing a query.

Algorithms. We implemented the following, all in Java: (a) RBSim (Section 4.1); (b) $\mathsf{Match}_{\mathsf{Opt}}$, an optimized version of the strong simulation algorithm [20], which *only* checks subgraphs within d_Q hops of v_p for query Q (d_Q is the diameter of Q, and v_p is the match of the personalized node of Q); (c) RBSub (Section 4.2); (d) $\mathsf{VF2}_{\mathsf{OPT}}$, the subgraph isomorphism algorithm of [11] optimized like $\mathsf{Match}_{\mathsf{Opt}}$; (e) RBReach (Section 5); (f) BFS that tests reachability by breadth-first search, and $\mathsf{BFS}_{\mathsf{OPT}}$, which compresses a graph first [12] and then runs BFS on the compressed graph; and (g) the reachability algorithm LM of [13] using landmark vectors. Note that $\mathsf{VF2}_{\mathsf{OPT}}$, $\mathsf{Match}_{\mathsf{Opt}}$ and BFS use our optimization.

Evaluation. We tested the impact of graph size $|G|$, query (set) size $|Q|$ and resource bound α (with $c = 1$) on (a) running time, and (b) accuracy. We adopted the accuracy measures given in Section 3 for pattern and reachability queries.

All the experiments were run on a machine powered by an Intel Core(TM) i7-3520M 2.90GHz CPU with 8GB of memory, using 64 bit Windows 7. Each experiment was run 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Graph patterns. The first set of experiments evaluated the accuracy, efficiency and scalability of (a) RBSim versus $\mathsf{Match}_{\mathsf{Opt}}$; and (b) RBSub versus $\mathsf{VF2}_{\mathsf{OPT}}$. We report the results for simulation and subgraph queries together, as they were tested in the same setting.

[3] *http://netsg.cs.sfu.ca/youtubedata/*

[4] *http://webscope.sandbox.yahoo.com/catalog.php?datatype=g*

Algorithms	*Youtube*			*Yahoo*		
	1.1	**1.6**	**2.0**	**1.1**	**1.6**	**2.0**
RBSim	7%	12%	19%	7%	14%	21%
RBSub	8%	15%	21%	8%	17%	24%

Table 2: The ratio of $\alpha|G|$ to $|G_{d_Q}(v_p)|$ ($\alpha \times 10^{-5}$)

Varying α. We first evaluated the impact of α using real-life graphs. Fixing $|Q| = (4, 8)$ (*i.e.,* $|V_p| = 4$ and $E_p| = 8$), we varied α from 0.0011% to 0.002%, in 0.0001% increments.

(1) Efficiency. We report the response time of the four algorithms in Figures 8(a) and 8(b) on *Youtube* and *Yahoo*, respectively. The results tell us the following: on average, (a) for simulation, RBSim takes only 24.4% and 18.8% of the running time of $\mathsf{Match}_{\mathsf{Opt}}$ on *Youtube* and *Yahoo*, respectively; (b) for subgraph queries, RBSub takes 16.7% and 14.4% of the time of $\mathsf{VF2}_{\mathsf{OPT}}$ on these two graphs; (c) the larger α is, the longer RBSim and RBSub take, but only slightly, since $|G_Q| = \alpha|G|$ gets larger when α increases; and (d) RBSim and RBSub are efficient: they took 2 and 5 seconds on *Yahoo*, respectively, even when $\alpha = 0.002\%$.

Moreover, our algorithms visit only a small part of the d_Q-neighborhood $G_{d_Q}(v_p)$ of v_p (see Table 2 for examples). On average, RBSim visits from 7% to 19% of $|G_{d_Q}(v_p)|$ on *Youtube*, and from 8% to 21% on *Yahoo*, when α ranges from 0.0011% to 0.002%; for RBSub, it is from 7% to 21% on *Youtube* and from 8% to 24% on *Yahoo*. This is why RBSim and RBSub outperform $\mathsf{Match}_{\mathsf{Opt}}$ and $\mathsf{VF2}_{\mathsf{OPT}}$, respectively.

These confirm that resource-bounded query answering indeed gives us the efficiency we need on real-life graphs.

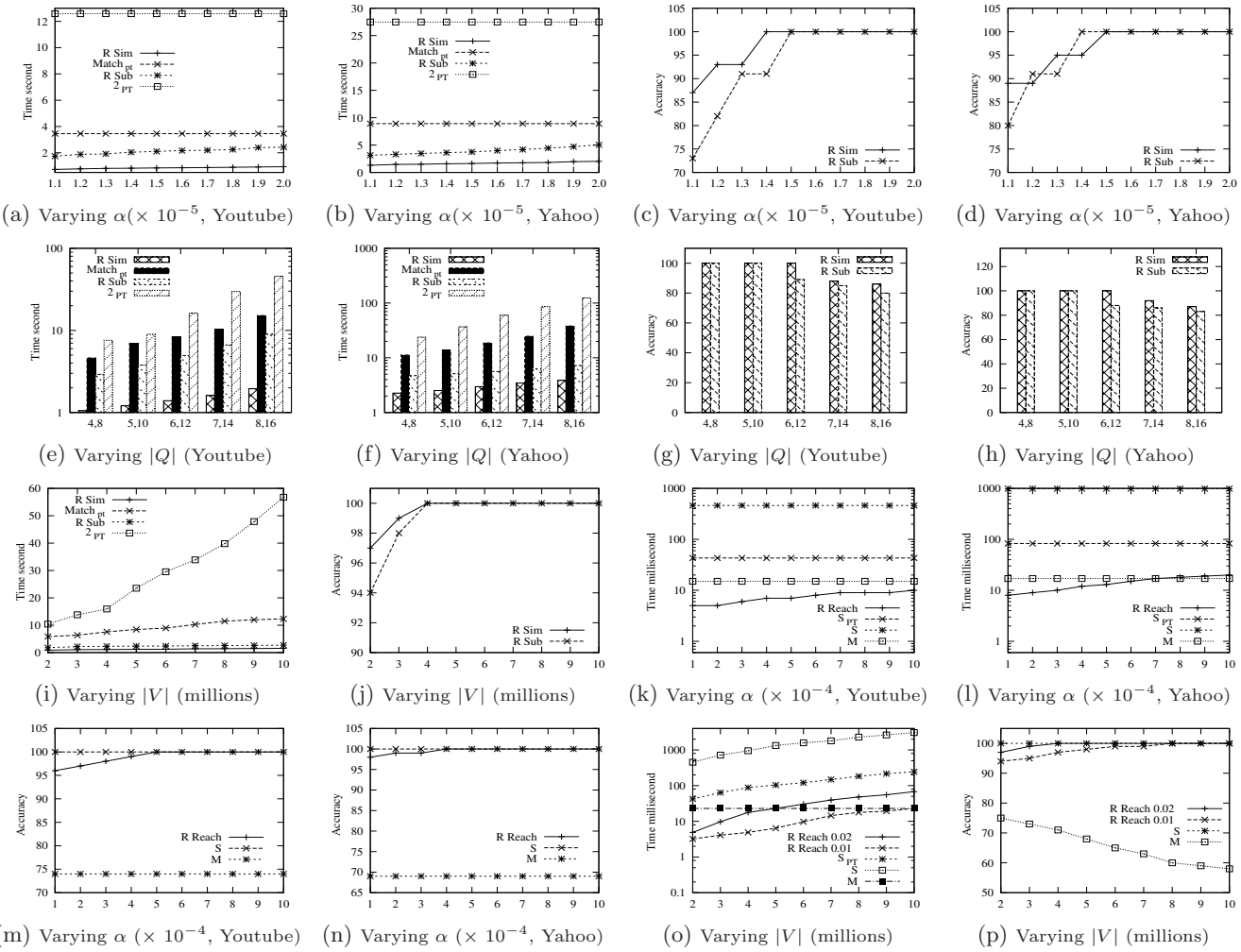
(2) Accuracy. In the same setting, we report the corresponding accuracy results in Figures 8(c) and 8(d) on *Youtube* and *Yahoo*, respectively. Note that $\mathsf{VF2}_{\mathsf{OPT}}$ and $\mathsf{Match}_{\mathsf{Opt}}$ are always 100% accurate and hence, are not shown.

We find the following. (a) Both RBSim and RBSub achieve high accuracy even when α is small. For example, the accuracy of RBSim ranges from 87% to 100% on *Youtube*, and 89% to 100% on *Yahoo*. (b) Better still, when $\alpha \geq 0.0015\%$, both RBSim and RBSub constantly get 100% accuracy. (c) When RBSim and RBSub achieve 100% accuracy, $|G_Q|$ is on average only 3% of the space bound induced by the theoretical minimum α given in Theorem 3(b), and it is between 17% and 19% of *the size of the d_Q-neighborhood of* v_p, respectively. (d) The larger α is, the higher the accuracy is, as expected, since G_Q can accommodate more information when α increases. These justify the effectiveness of resource-bounded query answering in practice.

Varying $|Q|$. We also evaluated the impact of $|Q|$. Fixing α as 0.01%, we varied $|Q|$ from $(4, 8)$ to $(8, 16)$.

(1) Efficiency. We report the efficiency of the algorithms on *Youtube* and *Yahoo* in Figures 8(e) and 8(f), respectively, which tell us the following. (a) The larger $|Q|$ is, the longer all these algorithms take. For RBSim, it takes $O(d_G|Q||G_Q|)$ time to find $G_Q = (V_{G_Q}, E_{G_Q})$ (Section 4.1), and $O(|Q||V_{G_Q}|(|V_{G_Q}|+|E_{G_Q}|))$ time to find matches in G_Q (Section 1). Hence the larger Q is, the longer it takes; similarly for RBSub. Nonetheless, RBSim and RBSub are less sensitive to $|Q|$ than $\mathsf{Match}_{\mathsf{Opt}}$ and $\mathsf{VF2}_{\mathsf{OPT}}$. (b) On average, RBSim and RBSub take 14.9% and 16.9% of running time of $\mathsf{Match}_{\mathsf{Opt}}$ and $\mathsf{VF2}_{\mathsf{OPT}}$, respectively. The improvement by our algorithms becomes more substantial for larger queries.

(2) Accuracy. Figures 8(g) and 8(h) report the accuracy results: (a) the larger $|Q|$ is, the lower the accuracy is for



(a) Varying $\alpha(\times 10^{-5}$, Youtube) (b) Varying $\alpha(\times 10^{-5}$, Yahoo) (c) Varying $\alpha(\times 10^{-5}$, Youtube) (d) Varying $\alpha(\times 10^{-5}$, Yahoo)

(e) Varying $|Q|$ (Youtube) (f) Varying $|Q|$ (Yahoo) (g) Varying $|Q|$ (Youtube) (h) Varying $|Q|$ (Yahoo)

(i) Varying $|V|$ (millions) (j) Varying $|V|$ (millions) (k) Varying α ($\times 10^{-4}$, Youtube) (l) Varying α ($\times 10^{-4}$, Yahoo)

(m) Varying α ($\times 10^{-4}$, Youtube) (n) Varying α ($\times 10^{-4}$, Yahoo) (o) Varying $|V|$ (millions) (p) Varying $|V|$ (millions)

Figure 8: Performance evaluation

RBSim and RBSub. This is because bounded resources allow us to access at most $\alpha|G|$ amount of data regardless of $|Q|$. (b) Nonetheless, the accuracy is above 86% for RBSim and above 80% for RBSub. Moreover, they achieve 100% for Q as large as $(5, 10)$. In practice, Q is typically small.

Varying $|G|$: Efficiency and accuracy. Fixing $|Q| = (4, 8)$ and $\alpha = 0.003\%$, we varied the node number $|V|$ of synthetic graphs from 2M to 10M, and set $|E| = 2|V|$. As shown in Fig. 8(i), (a) on average RBSim takes only 14.6% of the running time of $\mathsf{Match}_{\mathsf{opt}}$ and RBSub takes 13.8% of the time of $\mathsf{VF2}_{\mathsf{OPT}}$. (b) Both RBSim and RBSub scale well with $|G|$, and are much less sensitive to the change of $|G|$.

As shown in Fig. 8(j), (a) in all cases, the accuracy is above 97% for RBSim and 94% for RBSub, and mostly 100%; and (b) the larger $|V|$ is, *the more accurate* the algorithms are, due to the locality of pattern queries and our search strategy.

Exp-2: Reachability queries. This set of experiment evaluated the performance of our algorithm RBReach compared to BFS, $\mathsf{BFS}_{\mathsf{OPT}}$ and LM. We generated a set of 100 reachability queries, and report the average below. Following [13], we sampled $4 * \log|V|$ landmarks for LM.

Varying α: Efficiency and accuracy. Varying α from 0.01% to 0.1%, we report the response time of the algorithms on *Youtube* and *Yahoo* in Figures 8(k) and 8(l), respectively. The results show the following. (a) RBReach *substantially outperforms* BFS and $\mathsf{BFS}_{\mathsf{OPT}}$ in efficiency. It takes on average 1.6% and 17.4% of the running time of BFS and $\mathsf{BFS}_{\mathsf{OPT}}$, respectively. (b) When α increases, the running time of RBReach gets longer, as expected; but it is not very sensitive to α. (c) RBReach performs better than LM on *Youtube*. On *Yahoo*, LM does better when $\alpha > 0.07\%$. Nonetheless, as shown in Fig. 8(n), RBReach achieves 100% accuracy on *Yahoo* when $\alpha \leq 0.04\%$, when RBReach is faster than LM.

Moreover, RBReach is accurate. As shown in Figures 8(m) and 8(n), (a) in all cases, the accuracy is at least 96%, and is in general higher over denser graph *Yahoo*. (b) Moreover, when $\alpha \geq 0.05\%$, it is constantly 100% accurate! These verify that resource-bounded query answering is both efficient and accurate for non-localized reachability queries. The accuracy of LM, on the other hand, is from 69%–74%.

Varying $|G|$: Efficiency and accuracy. We varied $|V|$ of synthetic G from 2M to 10M (where $|E| = 2|V|$), and set α as 0.02% and 0.01%. Figure 8(o) tells us that RBReach scales well with $|G|$. (a) It is 58.8 and 5.2 times faster than BFS and $\mathsf{BFS}_{\mathsf{OPT}}$, respectively. (b) It outperforms LM when $|V| \leq 5$M for $\alpha = 0.02\%$; and is faster in all cases when α is small enough (*e.g.,* 0.01%), while the running time of LM is less sensitive to $|G|$ than RBReach. The accuracy of RBReach is above 97% (resp. 94%) for $\alpha = 0.02\%$ (resp. 0.01%) in all

cases (Fig. 8(p)). It increases with larger G, as the index $\mathcal{I}$ covers slightly more node pairs (with $|\mathcal{I}| \leq \alpha|G|$). In contrast, LM performs worse with larger $|V|$ as the number of landmarks sampled does not significantly increases.

Summary. We find the following. For patterns, (1) RBSim and RBSub are efficient: they are 5.5 times and 6.25 times faster than $\mathsf{Match}_{\mathsf{Opt}}$ and $\mathsf{VF2}_{\mathsf{OPT}}$, respectively, on real-life graphs; (2) they are accurate: when α is as small as 0.0015%, both achieve 100% accuracy; and (3) they scale well with $|G|$, without much performance degradation when G grows. The same holds on reachability queries: (4) RBReach is 62.5 and 5.7 times faster than BFS and $\mathsf{BFS}_{\mathsf{OPT}}$ on average, respectively, on real-life graphs; while its efficiency is comparable to that of LM, it is more accurate: 96%-100% vs. 69%-74%; (5) it gives us mostly exact answers when $\alpha \geq 0.05\%$; and (6) it scales well with $|G|$: when $|G|$ increases, so does its accuracy, without much penalty in efficiency. Finally, the tunable performance (controlled by α) of RBReach is more flexible than LM in balancing resource usage and accuracy.

7. CONCLUSION

We have proposed to query real-life graphs by resource-bounded query answering. We have studied its associated fundamental problems. We have also developed resource-bounded algorithms for answering localized (subgraph, simulation) and non-localized (reachability) queries. We have verified analytically and experimentally that these algorithms are able to efficiently find accurate approximate answers, even exact answers, with resource ratio α as small as 0.0015% for pattern queries, and 0.05% for reachability.

The study of resource-bounded query answering is still in its infancy. One topic is to explore resource-bounded algorithms for graph patterns without a personalized node. Another problem is to find, given a resource ratio α, *the maximum accuracy ratio* η that such algorithms can *guarantee*.

Acknowledgment. Fan, Wang and Wu are supported in part by 973 Programs 2014CB340302 and 2012CB316200, NSFC 61133002, Guangdong Innovative Research Team Program 2011D005, Shenzhen Peacock Program 1105100030834361, EPSRC EP/J015377/1 and NSF III 1302212.

8. REFERENCES

[1] L. A. Adamic and E. Adar. How to search a social network. *Social Networks*, 27(3):187–203, 2005.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[3] K. Bharat, A. Broder, J. Dean, and M. R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *J. Am. Soc. Inf. Sci.*, 51(12), 2000.

[4] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.

[5] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *ICDE Workshops*, 2011.

[6] J. D. Brutlag, H. Hutchinson, and M. Stone. User preference and search engine latency. In *Proc. ASA Joint Statistical Meetings 2008*, 2007.

[7] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'el, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *CIKM*, 2009.

[8] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: who is in your small world. *PVLDB*, 5(11), 2012.

[9] C. Chent, X. Yan, F. Zhu, and J. Han. gApprox: Mining frequent approximate patterns from a massive network. In *ICDM*, 2007.

[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5), 2003.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[12] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.

[13] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, 2010.

[14] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, 2005.

[15] R. Jain, D. Molnar, and Z. Ramzan. Towards a model of energy complexity for algorithms. In *WCNC*, 2005.

[16] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(1), 2013.

[17] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, 2009.

[18] R. E. Korf. Linear-time disk-based implicit graph search. *J. ACM*, 55(6):26:1–26:40, 2008.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[20] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *PVLDB*, 5(4), 2011.

[21] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[22] W. E. Moustafa, A. Deshpande, and L. Getoor. Ego-centric graph pattern census. In *ICDE*, 2012.

[23] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[24] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[25] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193–204, 1970.

[26] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[27] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[28] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[29] R. T. Stern, R. Puzis, and A. Felner. Potential search: A bounded-cost search algorithm. In *ICAPS*, 2011.

[30] J. T. Thayer and W. Ruml. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *ICAPS*, 2008.

[31] J. T. Thayer, R. Stern, A. Felner, and W. Ruml. Faster bounded-cost search using inadmissible estimates. In *ICAPS*, 2012.

[32] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, 2008.

[33] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[34] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

[35] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, 2013.

[36] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*. Springer, 2010.